Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Semi-Inflationary DATALOG: A Declarative Database Language with Procedural Features

A. Guzzo, D. Saccà

RT-ICAR-CS-04-7 **Aprile 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Semi-Inflationary DATALOG: A Declarative Database Language with Procedural Features

A. Guzzo[1], D. Saccà[1,2]

Rapporto Tecnico N.: **RT-ICAR-CS-04-7**

Data: **Aprile 2004**

---

[1] Università degli Studi della Calabria, DEIS, Via P. Bucci 41C, Rende (CS)
[2] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, Via P. Bucci 41C, 87036 Rende(CS)

Semi-Inflationary DATALOG: A Declarative Database Language with Procedural Features

Antonella Guzzo [a,*], and Domenico Saccà [a,b]
[a] *DEIS Dept., UNICAL, Rende, Italy.*
E-mail: {guzzo,sacca}@deis.unical.it

[b] *ICAR-CNR, Rende,Italy.*
E-mail: sacca@icar.cnr.it

This paper presents a rule-based database language which extends stratified DATALOG by adding a controlled form of inflationary fixpoint, immersed in a context of classical stratified negation with least fixpoint. The proposed language, called Semi-Inflationary DATALOG (DATALOG$^{@(*)}$ for short), smoothly combines the declarative purity of stratified negation with the procedural style of the inflationary fixpoint, DATALOG$^{@(*)}$ is particularly suitable to express algorithms in a mixed style: declarative rules, whenever it is natural and convenient, and procedural ones, any time it is easier to list the sequence of single actions. In the latter case, in order not to oblige the programmer to supply unnecessary procedural details, a number of choice constructs are available to express don't care non-determinism. The semantics of a DATALOG$^{@(*)}$ program is given using stable models by means of rule rewriting into a DATALOG program with choice and XY-stratification. In addition the complexity and expressive power of DATALOG$^{@(*)}$ queries is precisely characterized and some lights are put on the related class NQPTIME as well.

Keywords: *DATALOG, Stable model, Inflationary fixpoint, Non-Deterministic Query, Complexity*

1. Introduction

DATALOG is a rule-based declarative database language that is amenable to very efficient implementation as demonstrated by a number of prototypes of deductive database systems [10,11, 20,30]. Many proposals have been issued to extend DATALOG in order to support nonmonotonic queries, mainly by means of various forms of negation in the bodies of the rules. The first solution was *stratified negation* [5,9,35], which has a simple, intuitive semantics leading to efficient implementation. Unfortunately, this type of negation has a reduced expressive power for it can only express a proper subset of fixpoint queries.

The next step toward greater expressive power was to remove the condition that there is no recursion through negation. In this framework, a dramatic leap in expressive power is provided by the concept of *stable model* [12] but this gain is not without complications. Indeed the usage of unrestricted negation in programs is often neither simple nor intuitive, and, for example, might lead to writing programs that have no total stable models.

A great deal of research was focused on overcoming limitations of unstratified negation for nonmonotonic DATALOG and can be classified in two main directions:

1. several proposals have given up declarative semantics by falling back on procedural semantics, e.g. those based on the inflationary fixpoint computation procedure [2,3,23];
2. other proposals insisted in adopting a model-theoretic semantics without surrendering the naturalness and efficiency of stratified negation — they use a disciplined form of stable model semantics to refrain from abstruse forms of unstratified negation which may lead to undefinedness or unnecessary computational complexity.

As argued in [18], the core of a desirable database language in the latter approach should be stratified DATALOG, that is extended with only pre-

*Corresponding author: Antonella Guzzo, DEIS, University of Calabria, via P. Bucci 41C, 87036 Rende, Italy.

AI Communications
ISSN 0921-7126, IOS Press.

defined types of non-stratified negation, hardwired into ad-hoc constructs. A powerful construct for capturing a controlled form of unstratified negation is the *choice*, whose semantics was defined in terms of stable models in [31] by exploiting the nondeterminism implicit in the notion of such models. The combination of choice with extrema aggregates is investigated in [19] and many other facets of logic programming with choice are detailed in [14]. Recently, the problem of extending choice DATALOG to capture various complexity classes of database boolean queries and to express search and optimization queries has been addressed respectively in [18] and in [17].

Another interesting direction in extending the expressive power of DATALOG by a disciplined usage of unstratified negation is represented by XY-stratification which was first introduced in [39] and has later been used to model updated and active rules [37,38]. The recursive predicates of an XY-stratified program have a temporal argument which is used to enforce local stratification, a weak form of stratification introduced in [29]. It is interesting to observe that XY-stratification has a procedural style and, indeed, it can easily simulate inflationary fixpoint. In a sense XY-stratification represents a bridge between the two possible styles in DATALOG: procedural and declarative.

Our "current" belief (the quoted adjective is mandatory for believes) is that a declarative style should be adopted in all situations for which it is natural and one should switch to a procedural style as soon as it becomes more effective to list the sequence of actions to get the result. Possibly, to reduce the "rudeness" of procedurality, it may result convenient to use a certain dose of nondeterminism to hide some unnecessary details.

In this paper we present a new language, called Semi-Inflationary DATALOG (DATALOG$^{@(*)}$ for short), which smoothly combines the declarative purity of stratified negation and of least fixpoint with an inflationary fixpoint, suitably disciplined by means of the usage of ad-hoc predicates and rules. Choice constructs may occur only in inflationary rules and provide the power of don't-care non-determinism to reduce the number of details while defining the inflationary fixpoint stages. The semantics of the two components of language is given in the uniform context of stable model semantics by means of classical rewriting of the choice constructs and the use of XY-stratification to implement inflationary fixpoint.

DATALOG$^{@(*)}$ is particularly suitable to express algorithms in a mixed style: declarative, whenever it is natural and convenient, and procedural, any time it is necessary to break down the single actions at the desired level of details, regulated by an appropriate usage of choice. As illustrated in the paper, such a style is very effective to to describe greedy algorithms.

The relevance of this paper is not only the proposal of a new language: actually there have been so many proposals of extensions of DATALOG around that one would object that there is no need for yet another proposal. An important result of the paper is that it provides a precise characterization of the expressive power and complexity of the language and of the class of database queries NQPTIME as well, for DATALOG$^{@(*)}$ queries turn out to coincide with this class.

NQPTIME was first introduced in [2,4] to characterize the queries in a language with inflationary fixpoint augmented with a non deterministic construct, the witness. NQPTIME was originally defined as the class od all non-deterministic database transformations which can be computed by a nondeterministic Turing machine in polynomial time. Later, to remove some imprecision, this definition was refined in [24] by requiring that the for each input, every branch of a non-deterministic computation halts into an accepting state. At the best of our knowledge, this is the first time that the queries of a DATALOG language are proved to express the class NQPTIME according to the definition given in [24].

The paper is organized as follows. In Section 2, after having recalled preliminary notions on DATALOG and the basic concepts of choice and of XY-stratification, we illustrate the syntax and the stable model semantics of stratified DATALOG$^{@(*)}$ programs, which is given by rewriting the rules to eventually produce an XY-stratified programs with choice. In Section 3 we introduce non-deterministic database queries, discuss their complexity and prove the results about the class of stratified DATALOG$^{@(*)}$ queries. In Section 4 we present some extensions of DATALOG$^{@(*)}$ to express inflationary fixpoint in a simpler and more declarative way, thus providing a powerful formalism for describing greedy algorithms, as confirmed by a meaningful example. Finally we draw the conclusion and discuss further lines of research in Section 5.

2. The Semi-Inflationary language DATALOG$^{@(*)}$

2.1. Preliminaries on DATALOG, Choice and XY-Stratification

We assume that the reader is familiar with basic notions of relational databases, logic programming and DATALOG [1,7,25,34].

We are given a *universe* U (that is a countable set of constant symbols) and a countable set S of relation symbols with given finite arities. Let r be any relation symbol in S, say with arity k: a *tuple on* r is any element of U^k, a *relation on* r is any finite set R of tuples on r, and the set of all relations on r is denoted by $inst(r)$. A (*relational*) *database scheme* $\mathbf{D}$ is a tuple $\langle r_1, \ldots, r_m \rangle$ of different relation symbols. A (*relational*) *database* D on $\mathbf{D}$ is a tuple $\langle R_1, \ldots, R_m \rangle$, where for each i, $1 \leq i \leq m$, $R_i \in inst(r_i)$. The *active domain* of a database D, denoted by U_D, is the set of all constants occurring in D. The set of all databases on $\mathbf{D}$ is denoted by $inst(\mathbf{D})$.

A *logic program* P is a finite set of rules r of the form $H(r) \leftarrow B(r)$, where $H(r)$ is an atom (*head* of the rule) and $B(r)$ is a conjunction of literals (*body* of the rule). A rule with empty body is called a *fact*. The *ground instantiation* of P is denoted by $ground(P)$; the *Herbrand universe* and the *Herbrand base* of P are denoted by U_P and B_P, respectively.

Let an interpretation $I \subseteq B_P$ be given — with a little abuse of notation we sometimes see I as a set of facts. Given a predicate symbol r in P_D, $I(r)$ denotes the set $\{t : r(t) \in I\}$ — by seeing r also as a relation symbol, $I(r)$ is a relation. Moreover, $pos(P, I)$ denotes the positive logic program that is obtained from $ground(P)$ by (i) removing all rules r such that there exists a negative literal $\neg A$ in $B(r)$ and A is in I, and (ii) by removing all negative literals from the remaining rules. Finally, I is a (*total*) *stable model* [12] if $I = \mathbf{T}^{\infty}_{pos(P,I)}(\emptyset)$, that is the *least fixpoint* of the classical *immediate consequence transformation* for the positive program $pos(P, I)$.

Given a logic program P and two predicate symbols p and q, we write $p \rightarrow q$ if there exists a rule where q occurs in the head and there is a predicate in the body, say s, such that either $p = s$ or $p \rightarrow s$. P is *stratified* if for each p and q, if $q \rightarrow p$ holds in it then p does not occur negated in the body of any rule whose head predicate symbol is q, i.e. there is no recursion through negation. Stratified programs have a unique stable model which coincides with the *stratified model*, obtained by partitioning the program into an ordered number of suitable subprograms ('strata') and computing the fixpoints of every stratum in their order [5].

A DATALOG$^{\neg}$ program is a logic program with negation in the rule bodies but without functions symbols — if the program is stratified then it is called DATALOG$^{\neg s}$. Predicate symbols can be either extensional (i.e., defined by the facts of a database — *EDB predicate symbols*) or intensional (i.e., defined by the rules of the program — *IDB predicate symbols*). A DATALOG$^{\neg}$ program P has associated a relational database scheme $\mathbf{D}_P$, which consists of all EDB predicate symbols of P. Given a database D on $\mathbf{D}_P$, the tuples of D are seen as facts added to P; so P on D yields the following logic program $P_D = P \cup \{q(t). : q \in \mathbf{D}_P \wedge t \in D(q)\}$.

The complexity of computing a stable model of P_D is measured according to the *data complexity* approach of [8,36] for which the program is assumed to be constant while the database is variable. It is well known that computing the unique stable model of a DATALOG$^{\neg s}$ program P on a database D can be done in time polynomial on the size of D whereas it requires exponential time (unless $\mathcal{P} = \mathcal{NP}$) in case P is not stratified. Actually, in the latter case, deciding whether there exists a stable model or not is $\mathcal{NP}$-complete [26].

A disciplined form of unstratified negation is the *choice* construct, which is used to enforce functional dependency (FDs) constraints on rules of a logic program and to introduce a form of nondeterminism. The formal semantics of the choice can be given in terms of stable model semantics [31]. A rule r with choice constructs, called a *choice rule*, has the following general format:

$$r : A \leftarrow B(Z), choice((X_1), (Y_1)), \ \ldots, \\ choice((X_k), (Y_k)).$$

where, $B(Z)$ denotes the conjunction of all the literals in the body of r that are not choice constructs, and X_i, Y_i, Z, $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$. Each construct $choice((X_i), (Y_i))$ prescribes that the set of all consequences derived from r, say R, must respect the FD $X_i \rightarrow Y_i$. Let $FD_r = \{X_i \rightarrow Y_i | i = 1, ..., k\}$.

The formal semantics of choice is given in terms of stable models by replacing the above choice rule with the following rules:

1. Replace r with a rule r' (called *modified choice rule*) obtained by substituting the choice atoms with the atom $chosen_r(W)$:
   $r' : A \leftarrow B(Z),\ chosen_r(W).$
   where $W \subseteq Z$ is the list of all variables appearing in the choice goals, i.e. $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.
2. Add the new rule (called *chosen rule*)
   $chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$
3. For each $choice((X_i),(Y_i))$ $(1 \leq i \leq k)$, add the new rule *diffChoice rule*
   $diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$
   where (i) the list of variables W' is derived from W by replacing each $V \notin X_i$ with a new variable V' (e.g. by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $V \neq V'$, for some variable $V \in Y_i$ and its primed counterpart $V' \in Y_i'$.

Throughout the paper we shall use a simple variation of choice, denoted by ! (to be read "choiceAny" rather than "cut" as in the Prolog jargon), which nondeterministically selects one consequence. Thus this construct is a shorthand of $choice((),(Z))$, where Z is the list of all variables occurring in the rule body, according to the meaning of the FD $\emptyset \rightarrow Z$.

Another approach in disciplining unstratified negation is to add a distinguished *stage* (temporal) argument to recursive predicate symbols and to allow only two types of recursive rules:

1. *X-rule* when the stage argument of the head predicate is the same variable as in all stage arguments of the literals in the body which only occur positive;
2. *Y-rule* when the head stage argument is $T+1$ and all stge arguments of the literals in the body are equal to T, where T is a variable — in this case negation is allowed.

This extension, introduced in [39], is called *XY-Stratification* and has been used to model updated and active rules [37,38]. XY-stratified programs are indeed locally stratified [29] and, therefore, there exists a unique stable model although it can be infinite because of the temporal argument. Nevertheless, for practical applications it is possible to include restrictions into a XY-stratified program in order to guarantee both the finiteness of the stable model and its computation in polynomial time.

In the next section we shall present a language whose semantics is based on XY-stratification and on extended choices for which both existence and finiteness of stable models is guaranteed and one of them can be computed in polynomial time. We note that the combination of XY-stratification and choice has been first used in [6] to model various planning problems.

2.2. The Language DATALOG$^{@(*)}$

We next present *Semi-Inflationary* DATALOG, called DATALOG$^{@(*)}$ for short. In addition to classical EDB and IDB DATALOG predicate symbols, the language includes *inflationary* (*I-IDB*) predicate symbols which correspond to IDB relations that are computed by means of an *inflationary fixpoint*, i.e., all tuples computed at each iteration of the fixpoint are added to to the tuples computed at the previous iteration. Formally, given a logic program P and an interpretation I, the *inflationary immediate consequence transformation* is defined as $T_P(I) \cup I$ rather than just $T_P(I)$ as for classical least fixpoint. Inflationary fixpoint and inflationary DATALOG have been proposed in the literature (see for instance [2,3,23]). The novelty of DATALOG$^{@(*)}$ is that the two types of fixpoint are mixed together and can interact each other so that the programmer can switch from a declarative style to a procedural one by selecting the most suitable approach for each subproblem. The procedural style of inflationary fixpoint also assumes declarative flavor because non-deterministic constructs simplify the formalization and hide some low-level implementation details.

An I-IDB predicate has the format $p(X)@(S)$, where X is a list of arguments and S is the *stage argument* that denotes the stage of the inflationary fixpoint at which a tuple has been added to the relation p. A stage term $@(S)$ can have one of the following formats:

- $@(0)$ is the initial stage of an inflationary fixpoint;
- $@(.)$ denotes the current stage of an inflationary fixpoint;
- $@(+)$ denotes the next stage;
- $@(*)$ denotes all stages in an inflationary fixpoint (or up to the current stage if the fixpoint is not yet reached);
- $@(\hat{})$ denotes the last stage in an inflationary fixpoint that has been already computed.

In order to verify conditions at the various stages of an inflationary fixpoint, we use additional predicate symbols called *X-IDB* which have a stage argument as I-IDB symbols but of only two types: @(.) or @(0).

A $\texttt{DATALOG}^{@(*)}$ program consists of rules which may contain any type of EDB and IDB predicates in the body whereas I-IDB and X-IDB predicates are subject to some restrictions. In particular, a program comprises:

- *declarative rules* with IDB predicates in the heads; no X-IDB predicate symbol may occur in the body and possible I-IDB predicates in the body must have stage argument of the form either @(*) or @(^), i.e., I-IDB relations must be already computed before making a declarative rule inference;
- the *inflationary rules* having one of the following formats: $p(X)@(S) \leftarrow B, C_1, \cdots, C_s$, where X is a list of arguments, $s \geq 0$, C_i $(0 \leq i \leq s)$ are choice predicates and B is a conjunction of EDB, IDB and I-IDB literals. The inflationary rules are classified according to the format of the head iterative argument and have number of restrictions in the body:
  1. *initial rule*: the head predicate can be I-IDB or X-IDB with stage argument @(0); no I-IDB or X-IDB predicate may occur in the body; moreover, if the head predicate is I-IDB then the body may also contain choice predicates;
  2. *X-rule*: the head predicate is X-IDB with stage argument @(.), possible I-IDB predicates in the body must have stage arguments of the form @(.) or @(*); no choice predicates are allowed in the body;
  3. *Y-rule*: the head predicate is I-IDB with stage argument @(+), X-IDB predicates may occur in the body, and possible I-IDB predicates must have the same symbol as the head and stage argument @(.) or @(*); the body may contain choice predicates.

Example 1 Spanning Tree. *We are given two EDB predicate symbols* `arc` *and* `node`*, defined by a number of suitable facts, which encode an undirected graph, say G. We have two IDB-predicate symbol,* `path` *and* `disconnected` *to preliminary check whether the graph is connected or not. There is one I-IDB symbol,* `st`*, which allows to construct a spanning tree of G — a spanning tree is a subgraph of G where all nodes in G are reached from its root and no two of its arcs enter into the same node. An X-IDB predicate symbol* `reached` *is used to verify whether, at the current stage of the inflationary fixpoint, a node is reachable from the root. At beginning, (i.e. at the stage @(0)) a node in the graph is non-deterministically selected as root by means of the choice predicate* !*, provided that the graph is connected. After making this choice, the Y-rule is triggered in order to select the arcs to be included in the spanning tree:*

```
path(X,X)          ← node(X).
path(X,Y)          ← arc(X,Z), path(Z,Y).
disconnected()     ← node(X), node(Y), ¬path(X,Y).
st(root,X)@(0)     ← ¬disconnected(), node(X), !.
st(X,Y)@(+)        ← arc(X,Y), reached(X)@(.),
                     ¬reached(Y)@(.),
                     choice((Y),X).
reached(X)@(.)     ← st(_,X)@(*).
```

□

To simplify the notation, we use some syntactic sugar for writing I-IDB or X-IDB literals with safe negation (i.e., bound variables) in the rule bodies: $\neg a(X)@(S)$, where X is a list of arguments containing some anonymous variables, stands for $\neg a'(Y)@(.)$, where a' is a new X-IDB predicated symbol defined by the rule:

$$a'(Y)@(.) \leftarrow a(X)@(S).$$

where Y is the list of all variables in X that are not anonymous. A similar notation is used also for IDB or EDB negative literals.

Example 2 *A different solution for the spanning tree, using only one inflationary symbol and the above shorthand notation, is presented next — we only write inflationary rules as the declarative ones remain unchanged:*

```
st(root,X)@(0) ← ¬disconnected(), node(X), ! .
st(X,Y)@(+)    ← arc(X,Y), st(_ ,X)@(*),
                 ¬st(_,Y)@(*), choice((Y),X).
```

By rewriting the negated I-IDB predicate with anonymous variables we shall (almost) go back to the original program of Example 1. □

Example 3 Parity Query. *This problem consists in verifying whether the number of tuples in a relation is even or odd. Suppose that such tuples are stored as facts of an EDB predicate symbol* `r`*. We use an I-IDB predicate symbol* `p` *to select one tuple X of*

r at each stage and flipping I from 0 (even number of tuples) to 1 (odd number) and conversely.

$\mathtt{p(X,1)@(0)} \leftarrow \mathtt{r(X)}, !.$
$\mathtt{p(nil,0)@(0)} \leftarrow \neg\mathtt{r(\_)}.$
$\mathtt{p(X,1)@(+)} \leftarrow \mathtt{p(\_,0)@(.)}, \mathtt{r(X)}, \neg\mathtt{p(X,\_)@(*)}, !.$
$\mathtt{p(X,0)@(+)} \leftarrow \mathtt{p(\_,1)@(.)}, \mathtt{r(X)}, \neg\mathtt{p(X,\_)@(*)}, !.$
$\mathtt{parity()} \leftarrow \mathtt{p(\_,0)@(\hat{})}.$

Note that the shorthand notation introduced above is used for all negative literals, including the one of the second rule.

□

Example 4 Propositional Horn Clauses. *Given a positive propositional logic program, we want to find all propositional variables that are true in the minimal model. To this end, we store all variables as facts of the EDB predicate symbol* var*; moreover every clause* $\mathtt{r} : \mathtt{A} \leftarrow \mathtt{B_1}, ..., \mathtt{B_n}$, $n > 0$, *is represented by the facts* $\mathtt{r\_H(r,A)}$, $\mathtt{r\_B(r,B_1)}, ...,$ $\mathtt{r\_B(r,B_n)}$*; finally, if* $n = 0$ *then the clause is stored as* $\mathtt{fact(r,A)}$*. By means of the I-IDB predicate symbol* fired *we select at each stage a rule whose body is currently true and make true its head variable. We also keep track of the rule fired in the derivation of a propositional variable and, later, the IDB predicate symbol* true_var *drops out this argument.*

$\mathtt{inactive(R)@(.)} \leftarrow \mathtt{r\_B(R,X)}, \neg\mathtt{fired(\_,X)@(*)}.$
$\mathtt{fired(R,X)@(0)} \leftarrow \mathtt{fact(R,X)}.$
$\mathtt{fired(R,X)@(+)} \leftarrow \mathtt{r\_H(R,X)}, \neg\mathtt{fired(\_,X)@(*)}, \neg\mathtt{inactive(R)@(.)}, !.$
$\mathtt{true\_var(X)} \leftarrow \mathtt{fired(\_,X)@(*)}.$ □

Given a $\mathtt{DATALOG}^{@(*)}$ program P and a database D, the *standard version* $sv(P_D)$ of P_D is the choice XY-stratified program obtained from P_D by applying the following rewriting:

1. Replace every I-IDB or X-IDB predicate symbol p, say with arity n, with a new predicate symbol p' with arity $n+1$, and every predicate $p(X)@(S)$ in a rule r as follows:
   - $p(X)@(0)$ is substituted by $p'(X, 0)$;
   - $p(X)@(.)$ is substituted by $p'(X, I)$, where I is a new variable;
   - $p(X)@(+)$ is substituted by $p'(X, I')$ and the predicate $I' = I + 1$ is added to the body of r;
   - $p(X)@(*)$ is substituted as follows depending of the type of the rule r:
     (a) *r is an inflationary rule*: $p'(X, I')$ and the predicate $I' \leq I$ is added to the rule body;
     (b) *r is a declarative rule*: $p\_all(X)$, where p_all is a new IDB predicate symbol defined by the rule: $p\_all(X) \leftarrow p'(X, \_).$;
   - $p(X)@(\hat{})$ is substituted by $p\_last(X)$, where p_last is a new IDB predicate symbol defined by:
     $p\_last(X) \leftarrow p'(X, I), \neg non\_p\_last(I).$
     $non\_p\_last(I) \leftarrow p'(X, I'), I' > I.$
2. for each X-rule, say with head $p(X)@(.)$, if no positive I-IDB or X-IDB literal occurs in the body of r with stage argument $@(.)$, add the predicate $q'(\_, I)$ to the body of the rewritten rule, where $q(X)@(S)$ is the first I-IDB or X-IDB predicate occurring in the body of r — this predicate serves to make negation safe;
3. for each Y-rule, say with head $p(X)@(+)$, add the following literals to the body of the rewritten rule:
   - $\neg p''(X, I)$, to avoid that the same tuple is added at different stages, where p'' is a new predicate symbol with arity $n + 1$ defined by: $p''(X, I) \leftarrow p'(X, I'), I' \leq I.$
   - $p'(\_, I)$ if no positive I-IDB or X-IDB literal occurs in the body of r with stage argument $@(.)$ — as for X-rules, this predicate serves to make negations safe.
4. all choice predicates are rewritten in the standard way.

Example 5 *The standard version of the inflationary rules of the program in Example 1 is:*

$\mathtt{st'(root,X,0)} \leftarrow \neg\mathtt{disconnected()}, \mathtt{node(X)}, \mathtt{chosen_0(X)}.$
$\mathtt{st'(X,Y,I')} \leftarrow \neg\mathtt{st''(X,Y,I)}, \mathtt{I' = I + 1}, \mathtt{arc(X,Y)}, \mathtt{reached'(X,I)}, \neg\mathtt{reached'(Y,I)}, \mathtt{chosen_1(Y,X)}.$
$\mathtt{reached'(X,I)} \leftarrow \mathtt{st'(\_,\_,I)}, \mathtt{st'(\_,X,I')}, \mathtt{I' \leq I}.$
$\mathtt{st''(X,Y,I)} \leftarrow \mathtt{st'(X,Y,I')}, \mathtt{I' \leq I}.$
$\mathtt{chosen_0(X)} \leftarrow \neg\mathtt{disconnected()}, \mathtt{node(X)}, \neg\mathtt{diffchoice_0(X)}.$
$\mathtt{chosen_1(Y,X)} \leftarrow \neg\mathtt{st''(X,Y,I)}, \mathtt{I' = I + 1}, \mathtt{arc(X,Y)}, \mathtt{reached'(X,I')}, \neg\mathtt{reached'(Y,I)}, \neg\mathtt{diffchoice_1(Y,X)}.$
$\mathtt{diffchoice_0(X)} \leftarrow \mathtt{chosen_0(X')}, \mathtt{X \neq X'}.$
$\mathtt{diffchoice_1(Y,X)} \leftarrow \mathtt{chosen_1(Y,X')}, \mathtt{X \neq X'}.$

□

We next introduce the notion of stratification for DATALOG$^{@(*)}$ programs. Given a DATALOG$^{@(*)}$ program P, the *dependency graph* $G_P = (N, E)$ of P is a directed graph, where the nodes in N are all predicate symbols in P and an arc (q, p) is in E if there exists a rule r in (P) such that p is the head predicate symbol and q occurs in the body of r; moreover, the arc (q, p) is *labeled* with

'−' in the following cases:

1. q is an EDB or IDB predicate symbol and a predicate with this symbol occurs negated in the body of r — this is the classical stratification induced by negation;
2. r is a declarative rule, q is an I-IDB predicate symbol and a predicate with this symbol occurs (not necessarily negated) in the body of r — to contribute to the definition of an IDB predicates symbol, an I-IDB symbol must be at a lower stratum;
3. r is an X-rule, q is an X-IDB predicate symbol and a predicate with this symbol occurs negated in the body of r — recursion through negation is not allowed among X-IDB predicates;

'+' if r is an Y-rule and q is an X-IDB predicate symbol — recursion through negation is allowed in this case because of the stage increase.

Given an I-IDB predicate symbol p, the *X-Component* of p, denoted $XC(p)$, is recursively defined as: (i) p is in $XC(p)$, and (ii) if q is an X-IDB symbol and there exists a node $q' \in XC(p)$ such that either (q', q) or (q, q) is in G_P. We shall require that I-IDB symbols have disjoint X-components; on the other hand, X-IDB symbols not included in any X-component do not have any meaning.

A program P is stratified if the following two conditions hold:

1. no cycle in G_P contains some edge with label '-' and no edges with label '+';
2. for any two distinct I-IDB predicate symbols p and q, $XC(p) \cap XC(q) = \emptyset$.

Example 6 *The dependency graph of the program of Example 1 contains 2 cycles, both without '−' arcs: the loop on* path *and the cycle between* st *and* reached *having one arc with no label and the other arc with label '+'. So this program is stratified. Note that in this case the label '+' in the cycle does not play any role. To appreciate its relevance suppose that the last two rules in Example 1 are replaced by:*

$$\texttt{st(X,Y)@(+)} \leftarrow \texttt{arc(X,Y)}, \neg\texttt{unreached(X)@(.)}, \texttt{unreached(Y)@(.)}, \texttt{choice((Y),X)}.$$

$$\texttt{unreached(X)@(.)} \leftarrow \neg\texttt{st(\_,X)@(*)}.$$

This time the cycle between st *and* unreached *contains an arc with label '−' and the other one with label '+': so the latter arc preserves stratification.*

Note that also the programs of Examples 3 and 4 are stratified. To see an example of non-stratified program, replace the third rule of the program in Example 1 with the following rule:

$$\texttt{disconnected()} \leftarrow \texttt{node(X)}, \neg\texttt{st(\_,X)@(*)}.$$

We get a cycle between disconnected *and* st *with '−' arcs but without '+' arcs: so the program is not stratified.* □

Proposition 1 *Let P be a stratified* DATALOG$^{@(*)}$ *program. Then for each database D on $\mathcal{DB}_P$,*

1. *$sv(P_D)$ admits at least one stable model and every stable model is finite;*
2. *computing a stable model of $sv(P_D)$ can be done in time polynomial in the size of D and is P-hard;*
3. *if no choice predicates occur in the rules of P then $sv(P_D)$ admits exactly one stable model.*

Proof. (1) The rewriting of a stratified DATALOG$^{@(*)}$ program guarantees that $sv(P_D)$ is locally stratified. In fact, every cycle with negation that may occur in the dependency graph must include an I-IDB predicate symbol, say p. Let us now consider the ground instantiation $ground(sv(P_D))$ and every ground instantiation of the above cycle. The node p is instantiated by two distinct ground predicates, say with stage arguments i and $i+1$, resp. So the cycle is broken and non cycle with '−' arcs occur in the dependency graph of $ground(sv(P_D))$. Hence $ground(sv(P_D))$ is locally stratified and, then, it admits at least one stable model. The fact that all stable models of $ground(sv(P_D))$ are finite depends from the fact that the stage argument is increased only after at least a new tuple is added to some I-IDB relation; as the number of all possible such tuples is finite, all stable models are finite as

well. Finally, as $sv(P_D)$ and $ground(sv(P_D))$ have the same stable models, Part (1) is proved.

(2) A stable model of $sv(P_D)$ can be computed by dividing the program into a number of I-IDB strata, one for each I-IDB predicate symbol and, in turn, dividing each of such strata into classical strata as for stratified negation. It is easy to see that the computation is performed in a number of steps bounded by the size of D. Finally, to see that the computation is P-hard, observe that the program of Example 4 allows to solve the well-known P-complete problem of deciding whether a propositional variable is derived from a horn clause program whose rule bodies may contain more than one variable.

(3) If there are no choice constructs, the program is deterministic so there is at most one stable model; by Part (1) of this proposition, there exists at least one stable model. This concludes the proof. □

3. Complexity of DATALOG$^{@(*)}$

3.1. Preliminaries on Function Complexity

In analyzing expressiveness and complexity of stratified DATALOG$^{@(*)}$ queries, we shall be mainly referring to the following classes of languages: deterministic and non-deterministic polynomial time (denoted by P and $\mathcal{NP}$, resp.). The reader can refer to [21,28] for excellent sources of information on this subject. In addition we shall refer to complexity classes of functions — the source for this material is mainly [32,33]. Let us next recall some of the basic consepts.

Let $f : \Sigma^* \mapsto \Sigma^*$ be a partial multi-valued function. Let $f(x)$ stand for the set of possible outcomes (*results*) of the function f when applied to the input string x. Thus, we write $y \in f(x)$, if y is a value of f on the input string x. Define $dom(f) = \{x \mid \exists y(y \in f(x))\}$ and $graph(f) = \{\langle x, y\rangle \mid x \in dom(f),\ y \in f(x)\}$. If $x \notin dom(f)$, we will say that f is undefined at x. The function f is *total* if $dom(f) = \Sigma^*$.

A *transducer* is a (possibly, non-deterministic) Turing machine T on Σ with a read-only input tape, a read-write work tape, and a write-only output tape. There are two types of final states: *accepting* and *rejecting*. For any string $x \in \Sigma^*$, we say that T accepts x if there is a computation of T on x that ends into an accepting state. For each $x \in \Sigma^*$ accepted by T, we denote by $T(x)$ the set of all strings that are written by T on the output tape in its accepting computations on input string x. Any non-deterministic transducer T *computes* the partial multi-valued function f such that for each x, $f(x) = T(x)$ if x is accepted by T or otherwise f is undefined at x.

PF is the class of all single-valued functions computed by deterministic polynomial-time bounded transducers. The class NPMV is defined as the set of all multi-valued functions computed by non-deterministic polynomial-time bounded transducers. It is known that a function is in NPMV if and only if it is both polynomially balanced (i.e., for each x, the size of each result in $f(x)$ is polynomially bounded in the size of x) and $graph(f)$ is in $\mathcal{NP}$. The subclass of NPMV for which $graph(f)$ is in P is denoted by NPMV$_g$ — this class is also called FNP in the literature.

Given a multi-valued function f, it is often interesting to establish whether there exists a single-valued function f' which *refines* f, i.e., for each input x in $dom(f)$, f' returns one of the possible results in $f(x)$. Indeed, when asking for solving the search problem defined by a multi-valued function (e.g., find a minimal spanning tree of an input graph), we are usually interested in obtaining *one* of its solutions. Therefore, in the actual computation, the multi-valued function is eventually replaced by a refining, single-valued one.

Let us now recall the formal definition of refinement [32]. Given two partial multi-valued functions f and g, define g to be a *refinement* of f if $dom(g) = dom(f)$ and $graph(g) \subseteq graph(f)$. Let F and G be two classes of partial multi-valued functions. Let f be a partial multi-valued function. Following [32], we define $f \in_c G$ if G contains a refinement of f. Moreover, we define $F \subseteq_c G$ if, for all $f \in F$, $f \in_c G$. An important practical question is whether a result of a multivalued function can be efficiently computed by means of a polynomial-time, single-valued function or, more in general, which classes of multivalued functions are refined by PF.

3.2. Stratified DATALOG$^{@(*)}$ *Queries*

A (*database non-deterministic*) *query* $Q = \langle \mathbf{D}, g\rangle$, where $\mathbf{D}$ is a database scheme and g is a relation symbol not in $\mathbf{D}$, is a partial recursive

(i.e., computable), generic multi-valued function from $inst(\mathbf{D})$ to $inst(g)$; for each $D \in inst(\mathbf{D})$, the set of results $Q(D)$ is the *answer* of the query on D and each $G \in Q(D)$ is in $inst_{U_D}(g)$, thus the constants in a query result must also appear in the database. Genericity for Q means that for any D on $\mathbf{D}$ and for any isomorphism ρ on U, $Q(\rho(D)) = \rho(Q(D))$ [8,36]. Informally speaking, the answer of a query does not depend on the internal representation of the constants in the database. An important consequence of genericity is that Q is a polynomially balanced function. To see a query as a function, we assume a standard encoding of both the input database and the answer relations in the form of strings. Hence, in what follows, we shall freely talk about classes of queries being subsets of classes of functions.

A query $Q =< \mathbf{D}, g >$ will be said *deterministic* if it is single valued; if g has arity 0, the query Q is said to be *boolean*.

Classical complexity theory classifies languages on the basis of how difficult is to decide that a given input string belongs to the language. Deterministic search queries have been classified in a similar fashion, defining a recognition problem associated to them: given a deterministic query Q and a database D over a fixed scheme $\mathbf{D}$, the *query output tuple recognition problem* (QOT) for $Q(D)$ amounts to determining whether a given tuple t belongs to the unique result in $Q(D)$. The QOT is an appropriate tool for defining deterministic query complexity classes since the overall result relation can be constructed by iteratively checking the QOT on all tuples (whose number is polynomially bounded).

However, the QOT is not appropriate for classifying non-deterministic queries as two tuples may belong to two different results returned by the query on the same input and, as a consequence, does not allow to construct any of the query result relations. So a suitable approach for classifying non-deterministic queries is to generalize, to the framework of database queries, the so called *graph-recognition problem*, which is used for classifying the complexity of multi-valued functions[32]. This corresponds to adopt the *Query Output Relation* problem (QOR for short), which, given a non-deterministic query Q and a database $D \in inst(\mathbf{D})$, and a relation R, amounts to determining whether $R \in Q(D)$.

Let C be P or $\mathcal{NP}$. Then define the query class $NQ^{r}\text{-}C$ as the set of queries $Q \in \mathbf{NQ}$ such that the QOR for Q can be solved within C.

Observe that the queries in $NQ^{r}\text{-}\mathcal{NP}$ correspond to the generic functions of the class NPMV and those in $NQ^{r}\text{-}P$ correspond to the generic functions in NPMV_g. As pointed out in [24], there are queries in either classes which, at the current stage of knowledge, cannot be refined by a polynomial-time single-valued function. Let $(NQ^{r}\text{-}\mathcal{NP})^T$ denote the subclass of all queries in $NQ^{r}\text{-}\mathcal{NP}$ that are total.

Fact 1 [24]

1. $NQ^{r}\text{-}P \not\subseteq_c$ PF *and* $NQ^{r}\text{-}\mathcal{NP} \not\subseteq_c$ PF*, unless* $P = \mathcal{NP}$*;*
2. $(NQ^{r}\text{-}\mathcal{NP})^T \not\subseteq_c$ PF*, unless* $co\mathcal{NP} = \mathcal{NP}$*.* □

A stratified $\texttt{DATALOG}^{@(*)}$ query Q is of the form $\langle P, g\rangle$ where P is a stratified $\texttt{DATALOG}^{@(*)}$ program and g is an IDB or I-IDB predicate symbol of P, and represents the database non-deterministic query $\langle \mathbf{D}_P, g\rangle$ such that for each $D \in inst(P_{\mathbf{D}})$, $Q(D) = \{M(g) | M \text{ is a stable model of } sv(P_D)\}$. Let $\texttt{Q-DATALOG}^{@(*)}$ denote the class of all stratified $\texttt{DATALOG}^{@(*)}$ queries. From Proposition 1 we immediately derive that this class is refined by PF.

Corollary 1 $\texttt{Q-DATALOG}^{@(*)} \subseteq_c$ PF*.*

Proof. Let $Q = \langle P, g\rangle$ be a stratified $\texttt{DATALOG}^{@(*)}$ query and D be a database on $\mathbf{D}_P$. Then a result of $Q(D)$ can be obtained by computing a stable model M of $sv(P_D)$ and returning $M(g)$. By Part (2) of Proposition 1, M is computed in polynomial time by a single-valued function. □

As shown next, a severe drawback of a query in $\texttt{Q-DATALOG}^{@(*)}$ is that finding a result is done in polynomial-time but testing whether a given relation is a result is not (unless P=$\mathcal{NP}$). This contrasts with the typical situation in classical $\mathcal{NP}$-hard search problems: the hard part is finding a result rather than testing it. Let $(NQ^{r}\text{-}\mathcal{NP})^T$ denote the subclass of all queries in $NQ^{r}\text{-}\mathcal{NP}$ that are total.

Proposition 2

1. $\texttt{Q-DATALOG}^{@(*)} \subseteq (NQ^{r}\text{-}\mathcal{NP})^T$ *and the containment is strict unless* $P = \mathcal{NP} \cap co\mathcal{NP}$*;*
2. $\texttt{Q-DATALOG}^{@(*)} \not\subseteq NQ^{r}\text{-}P$ *unless* $P = \mathcal{NP}$*.*

Proof. (1) The fact that $\texttt{Q-DATALOG}^{@(*)} \subseteq NQ^{r}\mathcal{NP}$ can be shown as follows. Take any stratified $\texttt{DATALOG}^{@(*)}$ query $Q = \langle P, g\rangle$ and any database D on $\mathbf{D}_P$. Let G be any relation on g. We have that $G \in Q(D)$ iff there exists a stable model M of $sv(P_D)$ such that $G = M(g)$. We nondeterministically select an interpretation M of $sv(P_D)$ and verify in (deterministic) polynomial time whether M is a stable model and $G = M(g)$. So $\texttt{Q-DATALOG}^{@(*)} \subseteq NQ^{r}\mathcal{NP}$. To see that indeed $\texttt{Q-DATALOG}^{@(*)} \subseteq (NQ^{r}\mathcal{NP})^{T}$, observe that there exists at least one stable model M of $sv(P_D)$ by Part (1) of Proposition 1 and, then, $Q(D)$ is not empty for it contains at least $M(g)$. So Q is total and, hence, $\texttt{Q-DATALOG}^{@(*)} \subseteq (NQ^{r}\mathcal{NP})^{T}$. The fact that the containment is strict unless $P = \mathcal{NP} \cap co\mathcal{NP}$ derives from Part (2) of Fact 1.

(2) In order to prove this part, it is sufficient to show that $\texttt{Q-DATALOG}^{@(*)}$ is $\mathcal{NP}$-hard. Consider the $\mathcal{NP}$-complete problem of deciding whether a graph has a Hamiltonian path, i.e., there is a permutation $v_1, ..., v_n$ of all nodes of the graph such that $(v_1, v_2), ... (v_{n-1}, v_n)$ are arcs of the graph. Consider the following $\texttt{Q-DATALOG}^{@(*)}$ program P:

$\texttt{path}(\texttt{root}, \texttt{X})@(\texttt{0}) \leftarrow \texttt{node}(\texttt{X}), !.$
$\texttt{path}(\texttt{X}, \texttt{Y})@(+) \leftarrow \texttt{arc}(\texttt{X}, \texttt{Y}), \texttt{path}(\_, \texttt{X})@(.), \neg\texttt{path}(\texttt{Y}, \_)@(*), !$
$\texttt{non\_hp}() \leftarrow \texttt{node}(\texttt{X}), \neg\texttt{path}(\_, \texttt{X})@(*).$
$\texttt{is\_hp}(\texttt{0}) \leftarrow \texttt{non\_hp}().$
$\texttt{is\_hp}(\texttt{1}) \leftarrow \neg\texttt{non\_hp}().$

and the query $Q = \langle P, is\_hp\rangle$. Given a graph G encoded into a database D, G has a Hamiltonian path iff the relation $\{(1)\}$ belongs to $Q(D)$. This concludes the proof. □

Next we provide two interesting characterizations of $\texttt{Q-DATALOG}^{@(*)}$. To this end, we first recall the definition of the class of queries NQPTIME as given in [24]: a non-deterministic query Q is in NQPTIME if it can be computed by a polynomial-time transducer such that for each input, each branch of the transducer's computation halts into an accepting state. Note that NQPTIME was first introduced in [2,4] to characterize the queries in a language with inflationary fixpoint augmented with a non deterministic construct, the witness. We point out that some time in the literature NQPTIME is imprecisely defined as: 'the class of all non-deterministic database transformations (i.e., total queries) which can be computed by a non-deterministic Turing machine in polynomial time'. Thus NQPTIME could be confused with $(NQ^{r}\mathcal{NP})^{T}$.

Before giving our last result, we need an additional definition. A non deterministic query $Q\langle \mathbf{D}, g\rangle$ is *listable* if all results of Q can be computed by a polynomial delay (deterministic) algorithm (see [22,15]), i.e., for each $D \in \mathbf{D}$ with size $|D|$:

- the algorithm executes at most $pol(|D|)$ machine instructions before either producing the first results or halting;
- after any result it executes at most $pol(|D|)$ machine instructions before either producing the next results or halting.

Theorem 1

1. $\texttt{Q-DATALOG}^{@(*)} =$ NQPTIME*;*
2. *All queries in* $\texttt{Q-DATALOG}^{@(*)}$ *are listable.*

Proof.

(1) Let $Q = \langle P, g\rangle$ a stratified $\texttt{DATALOG}^{@(*)}$ query. Given a database D on $\mathbf{D}_P$, by Proposition 1 we can write a non-deterministic algorithm which computes the stable models of $sv(P_D)$. As the non-determinism consists in selecting some tuples during fixpoint computations and the fixpoint will be eventually reached independently from the selections made (don't care non-determinism), the algorithm can be easily encoded into a polynomial-time transducer T such that for each input, each branch of the transducer's computation halts into an accepting state. Let us now prove the reverse: given any query $Q = \langle \mathbf{D}, g\rangle$ computed by a transducer T with the above properties, there exists a query $Q' = \langle P, g\rangle$ such that $\mathbf{D} = \mathbf{D}_P$ and for each D on $\mathbf{D}$, $Q(D) = Q'(D)$. Let *pol* be a polynomial function such that, given D on $\mathbf{D}$, $pol(U_D) \geq S$, where U_D is the active domain of D and S is the maximum for all results of g in $Q(D)$ of the sum of the number of branches with non-deterministic moves and the number of tuples in a result - obviously such a function exists. Let k be the degree of the polynomial *pol*: then, by assuming and order for U_D, we can use the induced ordering on the cartesian product U_D^{k+2} to keep track of the order in which the non-deterministic move and the writing on the output tape are made. We also order all possible choices for the next move at each non-deterministic branch. We now construct a stratified $\texttt{DATALOG}^{@(*)}$ program P with $\mathbf{D}_P = \mathbf{D}$ as fol-

lows. We have two I-IDB predicate symbols. The first one is used to preliminarily select an ordering of U_D. The second one, say p, is used to store data only on the following three types of moves (*y-moves*): (1) non-deterministic move with or (2) without writing on the output tape, and (3) deterministic move with writing on the output tape. The predicate p has $k+5$ arguments: an argument for stating the type of y-move, an argument for the index of the chosen move (only for a y-move of type 1 or 2), an argument for storing the value written on the output tape (only for a y-move of type 1 or 3), and $k+2$ arguments for ordering the tuples being constructed. The inflationary Y-rule is:

$$p(I', V', T', U'_1, ..., U'_{k+2})@(+) \leftarrow p(I, V, T, U_1, ..., U_{k+2})@(.), q(I', V', T', U'_1, ..., U'_{k+2}@(.).)$$

The X-IDB predicate symbol q computes the data (including the ordering $(k+2)$-tuple) for the next y-move by consulting the data on the current y-move, by using the stage argument "@(.)", as well as the data on all previous y-moves, by using the stage argument "@(*)". The computation of data for the next y-move is obviously polynomial. So it can be realized by a suitable subprogram defining q which uses stratified negation and the ordering preliminarily generated — recall that stratified negation with a successor relation captures polynomial time exactly, as pointed out in [27]. The program is completed with a subprogram defining the query goal g (say with arity m) which extract from p all the tuples corresponding to y-move of type 1 and 3 and collect the 3rd arguments of m consecutive of such tuples. This operation is obviously polynomial and, then, can be implemented by stratified negation with ordering. Hence, for each D, $Q(D) = Q'(D)$.

(2) Let $Q = \langle P, g \rangle$ be a stratified $\texttt{DATALOG}^{@(*)}$ query. By part (1) of this theorem, Q can be computed by a polynomial-time transducer such that for each input, each branch of the transducer's computation halts into an accepting state. It is then easy to construct a polynomial delay algorithm using a backtracking approach. □

4. Extending the Choices in $\texttt{DATALOG}^{@(*)}$

The core of $\texttt{DATALOG}^{@(*)}$ is the coexistence of the declarative style of stratified negation with an inflationary fixpoint which construct a relation by selecting tuples at the different stages — the choice construct serves here to introduce the power of a controlled non-determinism in such a selection. An interesting extension of the language is to further increase the capability of making the selection while performing the inflationary fixpoint.

Example 7 Team building.

We are given projects, employees and skills represented by the following EDB facts:

```
project(P#, Priority, NTeam).
employee(E#, Skill#, SubSkill#, Salary, Sex).
requiredSkill(P#, Skill#, SubSkill).
```

Each project has a priority (measuring its relevance) and requires a number of employees, each with a distinct skill and, possibly, a sub-skill. Such skills must be granted by assigning the available employees with the wanted skills - fitting also the sub-skill is not mandatory although preferred. We have to set up a team of employees of all projects, if possible.

To solve the problem we use the I-IDB predicate inTeam(Project#, Employee#) *to encode the fact that an employee is enrolled in a project. Step by step, any employee candidate to be included into the project team is non deterministically chosen provided that the team is not yet fully staffed:*

```
inTeam(P, E)@(+) ← ¬staffed(P)@(.),
                   candidate(E, P)@(.), !.
```

The X-IDB predicate symbol staffed *checks whether a project is staffed at current stage with the wanted skills - the sub-skills are not taken into account.*

```
staffed(P)@(.)       ← project(P, _, _),
                       ¬missingSkill(P)@(.).
missingSkill(P)@(.)  ← requiredSkill(P, S, _),
                       ¬skillInP(S, P)@(.).
skillInP(S, P)@(.)   ← inTeam(P, E)@(*),
                       employee(E, S, _, _, _).
```

Moreover, we have the following X-rules for inferring the employees that are not currently involved into any project, and whose skills are still missing:

```
candidate(E, P)@(.) ← employee(E, S, _, _, _),
                      ¬inSomeTeam(E)@(.),
                      requiredSkill(P, S, _),
                      ¬skillInP(S, P)@(.).
inSomeTeam(E)(.)    ← project(P, _, _),
                      inTeam(P, E)@(*).
```

The program we have written is not satisfactory for we would like to perform the selections according to the following criteria, listed in order of their relevances:

1. *the projects should be staffed in order of their priorities to avoid that high priority projects will eventually result unstaffed because some critical skills have been assigned to other projects:*
2. *in selecting an employee to cover a skill, the one who also fits the sub-skill should be preferred;*
3. *if possible, at least the 50% of the employees in a team should be women;*
4. *the employees with lower salary should be preferred in order to reduce the overall cost of each project.* □

Next we introduce additional constructs for making choices inside Y-rules. We assume that the universe U is ordered and define two powerful variations of the choice: $choiceMin(C)$ and $choiceMax(C)$, where C is a single variable defined on an ordered domain, which select the consequences with respectively the minimal and the maximal value for C. Given a rule r with $choiceMin$, say

$$a(Y) \leftarrow B(Z),\ choiceMin((X), C).$$

where $B(Z)$ is a conjunction of literals, Z is the list of all variables occurring in B and B is a variable in Z, r is rewritten as

$$a'(Y, C) \leftarrow B(Z).$$
$$nonMin(C) \leftarrow a'(Y, C), a'(Y', C'), C' < C.$$
$$a(Y) \leftarrow a'(Y, C), \neg nonMin(C).$$

A rule with $choiceMax$ is rewritten in a similar way. We point out that both constructs are indeed deterministic. Note also that they are derived from two powerful variations of the choice described in [19]: $choiceLeast((X), C)$ and $choiceMost((X), C)$, where X is a list of variables occurring in the body, which select respectively the minimal and the maximal value for C, while enforcing the FD $X \rightarrow C$. The fact that we only allow the choice inside Y-rule avoids some problems with the semantics of such predicates that are reported in [19].

We also introduce an additional construct for making selection that is not disciplining another form of unstratified negation but it is simply a shorthand for a particular stratified negation. The construct is $possibly(D)$, where D is any literal: EDB, IDB, I-IDB or X-IDB. This predicate selects from all tuples being added at a stage of the inflationary fixpoint, those which satisfy D if any or, otherwise, all of them — thus if the literal D is not satisfied we want to accept all consequences rather than reject them. Given an Y-rule with *possibly*, say:

$$a(X)@(+) \leftarrow B(Z),\ possibly(D).$$

we perform the following rewriting:

$$a(X)@(+) \leftarrow B(Z),\ D.$$
$$a(X)@(+) \leftarrow B(Z),\ \neg D.$$

In using choice construct and their variations, we shall assume that if a Y-rule contains more than one choice constructs, then their selections are made in the order such construct occur in the body. Thus, given a Y-rule of the form:

$$p(X)@(+) \leftarrow B, C_1, ..., C_n.$$

where B is a conjunction of EDB, IDB, I-IDB and X-IDB literals and C_i, $1 \leq i \leq n$ are choice predicates or the constructs we have just introduced, the consequences of the rules will be first filtered by C_1, then the selected tuples will be further filtered by C_2 and so on up to the final selection of C_n. We enforce the ordering of choices by a suitable rewriting. In particular, after rewriting the I-IDB and X-IDB predicates, we introduce $n-1$ further new predicate symbols in addition to p': $p'_1, ..., p'_{n-1}$. Then we write n copies of the Y-rule:

$$p'_1(X)@(+) \leftarrow B, C'_1.$$
$$p'_2(X)@(+) \leftarrow p'_1(X), B, C'_2.$$
$$\ldots$$
$$p'_{n-1}(X)@(+) \leftarrow p'_{n-2}(X), B, C'_{n-1}.$$
$$p'(X)@(+) \leftarrow p'_{n-1}(X), B, C'_n.$$

Finally, we allow aggregate predicates $count\{X : q(Y)@(.)\}$ or $count\{X : p(Y)@(.)\}$ to occur into the body of a Y-rule or and X-rule — in the latter case we require that q is not recursive with the head predicate symbol. Aggregate predicates are used as terms into comparison predicates: e.g., $N = count\{X : q(Y)@(.)\}$.

Example 8 Team Building with Extended Selection.

The Y-rule becomes:

$\texttt{inTeam}(\texttt{P}, \texttt{E})@(+)$

$\leftarrow$ project(P, Prty, _), $\neg$staffed(P)@(.),
candidate(E, P),
employee(E, Sk, SubSk, Sal, Sex),
choiceMax(Prty),
possibly(requiredSkill(P, Sk, SubSk))
possibly(equalOpp(P, Sex)@(.)),
choiceMin(Sal) !.

The order of selection constructs correspond to the relevance we assign to the various properties: first the priority of the project, then the availability of skills and, if possible, of the sub-skill; later on, the equal opportunity, then the minimal cost and finally only one of the remaining consequences is selected.

The X-IDB predicate symbol is defined by:

equalOpp(P, f)@(.) $\leftarrow$ lessWomen(P)@(.).
equalOpp(P, Sex)@(.) $\leftarrow$ project(P, _, _),
$\neg$lessWomen(P)@(.).
lessWomen(P)@(.) $\leftarrow$ project(P, _, Nteam),
count{E : women(P, E)@(.)}
$<$ Nteam $*$ 0.5.
women(P, E)@(.) $\leftarrow$ inTeam(P, E)@(.),
employee(E, _, _, f).

□

5. Conclusion

In this paper we have presented an extension of DATALOG called DATALOG$^{@(*)}$, which combines least fixpoint on stratified negation and inflationary fixpoint augmented with choice constructs. This language is particularly suitable to express algorithms in a mixed style: declarative rules interleaved with inflationary ones that are used any time it becomes easier to just list sequences of single actions rather than providing a complex declarative definition. The complexity complexity and expressive power of DATALOG$^{@(*)}$ queries have been precisely characterized and some lights are put on the related class NQPTIME as well.

In the paper we have also sketched some extensions to the choice constructs of DATALOG$^{@(*)}$ to express inflationary fixpoint in a simpler and more declarative way, thus providing a powerful formalism for describing greedy algorithms. By further exploiting the idea of adding procedural features to a declarative language, on-going research [16] is devoted to defining an extension of DATALOG to express events and nondeterministic state transitions, by using choice constructs to model uncertainty in dynamic rules. The proposed language, called Event Choice DATALOG provides a powerful mechanism to formulate queries on the evolution of a knowledge base, given a sequence of events envisioned to occur in the future. A distinguished feature of this language is the use of multiple spatio-temporal dimensions in order to model a finer control of evolution for a large class of problems, such as those involving planning tasks and workflow systems.

References

[1] Abiteboul, S., Hull, R., and Vianu, V., *Foundations of Databases*. Addison-Wesley. 1994.

[2] Abiteboul, S., E. Simon, E., and V. Vianu, V., Non-Deterministic Languages to Express Deterministic Transformations In *Proc. of the Ninth ACM PODS Conference*, pages 215–229, 1990.

[3] Abiteboul, S. and V. Vianu, V., Datalog Extensions for Databases Queries and Updates. *Journal of Computer and System Sciences*, 43, pages 62–124, 1991.

[4] S. Abiteboul and V. Vianu. Non-determinism in logic-based languages, *Annals on Mathematics and AI*, Vol. 3, No. II-IV, 1991.

[5] Apt, K., Blair, H., and Walker, A., Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kauffman, Los Altos, USA, 1988, 89-142.

[6] Brogi, A., Subrahmanian, V. S., and Zaniolo, C., Modeling Sequential and Parallel Plans. *Journal of Artificial Intelligence and Mathematics*, Vol.19, n3, April 1997.

[7] S. Ceri, G. Gottlob and L. Tanca, *Logic Programming and Databases*, Springer-Verlag, 1990.

[8] Chandra, A., and Harel, D., Structure and Complexity of Relational Queries. In *Journal of Computer and System Science*, Vol. 25, n1, pp. 99–128, 1982.

[9] Chandra, A., and Harel, D., Horn Clauses Queries and Generalizations. In *Journal of Logic Programming*, Vol. 25, n1, pp. 1–15, 1985.

[10] Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., and Zaniolo, C., The LDL System Prototype. In *IEEE TKDE*, Vol. 2, n1, pp. 76–90, 1990.

[11] Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., The KR system dlv: Progress report, comparison and benchmarks.In Proc. Sixth Int. Conf. on Priciples of Knowledge Representation and Reasoning (KR98),A.G.Cohn, L.Schubert, and S.Shapiro, Eds, Morgan Kaufmann Publishers, 406-417.

[12] Gelfond, M., and Lifschitz, V., The Stable Model Semantics for Logic Programming. *Proc. 5th Int. Conf. on Logic Programming*, pp.1070-1080, 1988.

[13] Gelfond, M., and Lifschitz, V., Action languages. *Electronic Transactions on Artificial intelligence*, Vol. 2, n 3-4, pp.193-210, 1998.

[14] Giannotti, F., Pedreschi, D., and Zaniolo, C., Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases. *Journal of Computer and System Sciences, 62*, 1, pp. 15–42, 2001.

[15] L.A. Goldberg. Listing graphs that satisfy first order sentences. *Journal of Computer and System Sciences*, 49, pp. 408-424, 1994.

[16] Greco, G., Guzzo A., Saccà, D., and Scarcello, F., Event Choice Datalog: A Logic Programming Language for Reasoning in Multiple Dimensions. In *Proc. of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04*, Verona, August 24–26, 2004.

[17] Greco, S., and Saccà, Search and Optimization Algorithm in Datalog. In *Computational Logic: From Logic Programming into the Future*, Kakas A., Sadri F. (edrs), Springer Verlag, 2002.

[18] Greco, S., Saccà, D., and Zaniolo, C., Extending Stratified Datalog to Capture Complexity Classes Ranging from P to QH. In *Acta Informatica*, Vol. 37 N10, pp 699-725, July 2001.

[19] Greco, S., and Zaniolo, C., Greedy Algorithms in Datalog. in *Theory and Practice of Logic Programming*, Vol. 1 N4, pp 381-407, July 2001.

[20] Ilkka Niemelä, Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Journal of Artificial Intelligence and Mathematics*, Vol.25, N3-4, 1999.

[21] Johnson, D. S., A Catalog of Complexity Classes. In *Handbook of Theoretical Computer Science*, Vol. 1, J. van Leewen (ed.), North-Holland, 1990.

[22] D. S. Johnson, M. Yannakakis, and C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27, pp.119–123, 1988.

[23] Kolaitis, P., and Papadimitriou, C., Why not Negation by Fixpoint? *Journal of Computer and System Sciences*, 43, pages 125–144, 1991.

[24] N: Leone, L: Palopoli, D. Sacca', On the Complexity of Search Queries, in T. Polle, T. Ripke, K.D. Schewe, eds., *Fundamentals of Information Systems*, pp.113–127, Kluwer Academic Publishers 1999.

[25] Lloyd, J., *Foundations of Logic Programming*. Springer-Verlag, 1987.

[26] Marek, W., Truszczynski, M., Autoepistemic Logic. *Journal of the ACM*, Vol. 38, No. 3, pp. 588–619, 1991.

[27] Papadimitriou, C., A Note on the Expressive Power of Prolog. In *Bull of the EATCS*, 26, pages 21–23, 1985.

[28] Papadimitriou, C. H., *Computational Complexity*. Addison-Wesley, Reading, MA, USA, 1994.

[29] Przymusinski T.C., On the Declarative and procedural Semantics of Stratified Deductive Databases, in *Foundations of Deductive Databases and Logic Programming*, (J.W. Minker, ed.), pp. 193-216, Morgan Kaufman, USA, 1988.

[30] Ramakrisnhan, R., Srivastava, D., and Sudanshan, S., CORAL — Control, Relations and Logic. In *Proc. of 18th Conf. on Very Large Data Bases*, pp. 238-250, 1992.

[31] Saccà, D., and Zaniolo, C., Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. ACM Symp. on Principles of Database Systems*, 1990, pp. 205-218.

[32] A. Selman, A taxonomy of complexity classes of functions, *JCSS*, 48, 357–381, 1994.

[33] A. Selman, Much ado about functions, *Proc. Conf. on Structures in Complexity Theory*, IEEE, 1996, 198–212.

[34] Ullman, J. K., *Principles of Data and Knowledge-Base Systems*, Vol.1-2. Computer Science Press, New York, 1988.

[35] Van Gelder, A., Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, Vol. 6, n. 1, pp. 109–133, 1989.

[36] Vardi, M., The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pp. 137–146, 1982.

[37] Zaniolo, C., Transaction-Conscious Stable Model Semantics for Active Database Rules. In *Proc. Int. Conf. on Deductive Object-Oriented Databases*, 1995.

[38] Zaniolo, C., Active Database Rules with Transaction-Conscious Stable Model Semantics. In *Proc. of the Conf. on Deductive Object-Oriented Databases*, pp.55–72, LNCS 1013, Singapore, December 1995.

[39] Zaniolo, C., Arni, N., and Ong, K., Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach, *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases*, 1993.